Ultracomputer Note #27 Revised February 1982

Molecular Simulation of Equilibrium Properties. Parallel Implementation

MALVIN H. KALOS
GABI LESHEM
B. D. LUBACHEVSKY

Computer Science Dept.
New York University
251 Mercer St., New York, N.Y. 10012

Abstract - A Monte Carlo algorithm for sampling representative configurations of a classical many-body system in equilibrium was implemented as a program for an asyncronous shared memory parallel computer. The algorithm is the iterative repetition of a cycle involving serial initialization and termination phases and a dominant computation in the middle that is suitable for parallelization. The efficiency of the parallel implementation is increased by allowing more than one such a cycle to be treated concurrently. Two versions of the implementation are discussed. The first one uses only one parallel feature, basic indivisible operation Fetch-And-Add [3, 4], and has necessary service subroutines built in. The second one employs primitives Request, Release and Suspend of the parallel operating system suggested in [2]. Timing analysis of simulated executions predicts high efficiency of this algorithm for parallel computers that include up to thousands of processing elements.

1. INTRODUCTION

A Monte Carlo algorithm [1] for sampling representative configurations of a classical many-body system in equilibrium was implemented as a program for the ultracomputer [5], an asynchronous shared memory parallel processor. The Monte Carlo method is often used in the study of condensed matter and would be practical for simulating fluids composed of complex molecules (e.g. water, hexane and larger molecules) if appropriate computing power were available. It is therefore a suitable and important example to study. Moreover it serves as a test vehicle for certain issues of program organization, as we shall indicate.

Like many Monte Carlo algorithms, the sampling procedure has the property that at one extreme (in this case ca. 1000 atoms altogether), a simple and effective way of organizing a parallel computation would be to assign a statistically independent replica of the system to each processing element (PE). One would learn rather little by simulating such a parallel program, except for the important ratio of global memory references per operation. It is easy, however, to show that in the worst case, there could be no more than one global fetch per six instructions. Stores of public data are relatively rare in this simple case.

As will be indicated below, the algorithm chosen involves two stages which are inherently serial and a third which is suitable to distribute among many processors.

The choice of model of the physical system involves a decision about the nature of the particles, specifically the number and character of external and internal degrees of freedom, the number of particles, and the boundary conditions. The model also includes a description of the internal forces - or more to the point - the potential energy acting between particles. For simplicity we treat point particles with pair-wise additive central forces. That means that the total potential energy of a configuration in which particle i is at position r(i) is:

$$V(r(1),\ldots,r(N)) = \sum_{i<j} v(|r(i)-r(j)|) = \sum_{i<j} v(i,j)$$

In the algorithm, one particle, say the k-th, is selected at random, and moved a random amount from r(k) to r′(k). The change in total potential energy :

$$V'-V = \sum_{i} v(|r(i)-r'(k)|) - v(|r(i)-r(k)|)$$

is computed and used as the basis for deciding whether or not the move is accepted (in the latter case, particle k is restored to its original position r(k)).

The computation of V′-V is normally the overwhelming computation in such an algorithm. Clearly the separate terms of the sum may be carried out in separate processors.

The computation of potential change is so dominant that various stratagems are employed to shorten the time involved. The most common for short range potentials is to approximate the function v(r) by one that is zero beyond some value, r_c. Then fewer terms in the sum are

needed provided one knows which particles are neighbors (i.e. which lie within a distance r_c) of each particle. A scheme which is particularly effective for large number of particles divides the region available into (usually) equal subregions and records the particles in each such subregion. Then, for any k, the sum required for V'-V is arranged as a sum over subregions near r(k) and an inner sum over the particles in each subregion. When a particle move is accepted, the particle may move to another subregion and the set of particles in each subregion must be updated.

In preparing the test specifications for this algorithm we aimed to keep faithfully the real complications as described above, while minimizing the non-essential programming associated with a large system of complex particles. Accordingly we took a system of N identical particles distributed around a ring (i.e. a one-dimensional system with periodic boundary conditions) and assumed a simple form for the potential, namely:

$$v(r) = \begin{cases} v_0[(s/r)^{12}-(s/r)^6], & \text{if } r \leq r_c; \\ 0, & \text{if } r > r_c. \end{cases}$$

where v_0 and s are used respectively to scale energies and lengths. The subregions were taken to be intervals of size approximately equal to r_c so that neighbors to be considered lie in three consecutive intervals.

In addition to updating and averaging V, the program also prepares a histogram of the distribution of the distances between the particles.

The k-th element of the histogram is the average of the following function

$$g_k(r(1),\ldots,r(N)) = \sum_{i<j} \chi_k(r(i), r(j))$$

where $\chi_k(r', r'') = 1$, if $L\times(k-1)/Nh \leq |r' - r''| < L\times k/Nh$; and $\chi_k(r', r'') = 0$, otherwise; $k = 1,\ldots,Nh$; Nh and L are parameters.

We would like here to point out briefly how our model problem is in fact a pessimistic case as compared with one that might truly be run on an ultracomputer (i.e. efficiency would be higher for a real problem). Later in discussing the significance of timing tests, we will treat this issue in greater detail.

Normally the computation of v(i,j) would be much more complicated. Either the functional form would be more elaborate, or for complex molecules, each v(i,j) would itself be a sum of terms over all pairs of atoms in the two molecules, and this sum would be assigned to one or a few processors. Also in two and especially three dimensions, the sum over neighboring subregions includes more terms. Both of the considerations increase the completely parallelizable part of the algorithm.

As will be seen below, we have used a simple scheme for leapfrogging two successive indivisible parts of the algorithm; this would be still effective (although less needed) in a more complex problem. In our model the particles are distributed on a ring and can move only in one dimension. Nevertheless, the program architecture and

the data-structure used to determine neighbors were designed to handle the multidimensional case as well.

2. PARALLEL IMPLEMENTATION

In our implementation we associated with each subregion a bin, i.e. a sequentially allocated list containing pointers to each particle in the subregion. If the list overflows, a linked-list is established, but the length of the sequential lists is chosen such that overflow is very unlikely.

At any step a particle may be moved from bin to bin, but this move can effect only the neighboring bins. Representing each bin by a sequential list instead of a linked list enables the distribution of the particles to processes to be carried out in parallel.

Figure 2.1 shows the structure of the data base in the public memory. There are 9 particles in 3 bins of width 30. Since there are 6 particles in the bin representing interval 60-90, but room for only 4, the presence of particles 5 and 4 are indicated by pointers.

The program proceeds by cyclicly executing the following 3 steps.

1. Generate: Pick a particle at random and move it a random amount. This is called a "try" and is accomplished by procedure "generate".

2. Calculate: Calculate the change in potential energy and histogram caused by the the moved particle. Only neighbors need be examined and they may all be done in parallel. This parallel computation is accomplished by procedure "pwork".

3. Summary: Check if the move is accepted, and update the data-base, the average energy and histogram accordingly. Updating is accomplished by procedure "update".

particle number	coordinate	linked-lists	comments
1	32	*	
2	75	*	
3	50	*	
4	62	nil	end of the linked-list
			extension for bin 3
5	68	4	pointer to particle 4
6	35	*	
7	83	*	
8	82	*	
9	89	*	

bin number	particles in bin	bin pointer	comments
1	* * * *	0	empty bin
2	3 1 6 *	-3	only 3 elements in the bin; the negative sign shows that 3 is not a pointer
3	9 7 8 2	5	bin overflow, 5 is the pointer to the first element on the linked-list for this bin

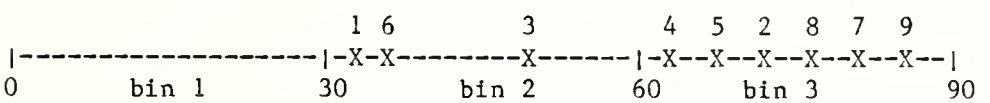


Figure 2.1. Example of the data base (9 particles in 3 bins)

Steps 1 and 3 are serial, so each is executed by one processor. Step 2 can obviously be parallelized by requesting a processor for the calculation of the change due to each neighbor. In a more realistic program this calculation would be much more elaborate and would consume most of the computing time, thus justifying the use of many processors.

One may examine more than one move at one time. To avoid excessive logical complexity in our test program, we limited the number of possible concurrent tries to two. It would in some cases improve the efficiency to allow more.

Two versions of the program were implemented. The first one was to be run using the WASHCLOTH-simulator [6] and made use essentially of the only one parallel feature, namely the operation Fetch-And-Add.‡

In this first version we tried to simulate some of the features of the MOP operating system [2] built in the program. We shall call this program the "pseudo operating system" version (POP version). The second version of the program (MOP version) was to be run using an interface only with MOP. MOP handles a queue of processes with multiplicity and priority, and provides the basic Request, Release and Suspend primitives. This can be accomplished by maintaining a number of (possibly large) circular arrays in public memory, as shown in [3].

For our special problem we need only one circular array of length 4, and a specific synchronization procedure to prevent overlapping of different execution cycles. Using this array we implemented a REQUEST

‡Fetch-And-Add (x, e) is the function-with-side-effect that returns the old value of x and replaces x by x + e. The Fetch-And-Add operation is indivisible in the sense that the result of concurently executed Fetch-And-Add operations is the same as if the operations were executed in some unspecified serial order (see [2, 5]). Note that the WASHCLOTH provides another parallel feature, function #pe - the processor identification number, (#pe = 0, 1,...npe-1, where npe is the total number of PEs in the system). However were it necessary this function could also be simulated using only Fetch-And-Add.

subroutine that is analogous to the Request primitive in MOP [2]. This subroutine is used to request a task with the desired multiplicity and a parameter list. The request is stored in the circular array, with the number of tasks to be executed, and can be accessed in a parallel fashion by all the processors. REQUEST is activated by the sequence:

CALL REQUEST(task number,multiplicity,parameter list)

Our program needs two priority levels: a high priority (= 2) for serial tasks, and a low priority (= 1) for the parallel tasks. Note that in the POP version the priority parameter is not maintained explicitly, but rather is implied in the structure of the program.

A busy-wait loop is used to suspend the execution of a process until the appropriate condition is fulfilled. This is accomplished in the POP version by the following subroutine SYNCH (K), which suspends the beginning of the work on #try=n until the work on all tries up to n-K is completed (synchronization K steps back). Note that K = 0 reduces to the usual synchronization where only one task can be examined at any time. In our implementation K = 1.

```
PROCEDURE SYNCH (K)
   IF FetchAndAdd (nbusyp(index) , 1) < npe THEN
      $ busy wait loop until previous tasks ready
      WAIT UNTIL(nbusyp((index - K) MOD (2×K+2)) = 0)
   ELSE nbusyp(index) := 0
   ENDIF
   index := (index + 1) MOD (2×K+2)
END
```

where: index is initialized to 1 in all the processors;

npe is the number of processors that should be synchronized;

nbusyp is an array of length 2×k+2, counting the number of busy processors in each step of the computation; each element of nbusyp is initialized to 1.

Now we present a schematic of the MOP version of our program. Note that in the MOP version an invocation of system primitive Suspend covers the functions of the subroutine SYNCH (K) in the POP version.

The change in the energy is calculated as the difference between the energy of the moved particle at its old position and its energy at its new position. Paramlist+ is the parameter list for the calculation of the energy in the new position; paramlist- is for the calculation of the energy in the old position. The parameter try# indicates which generate-calculate-summary execution cycle is being attempted.

The main routine of the program for #try >= 1 is:

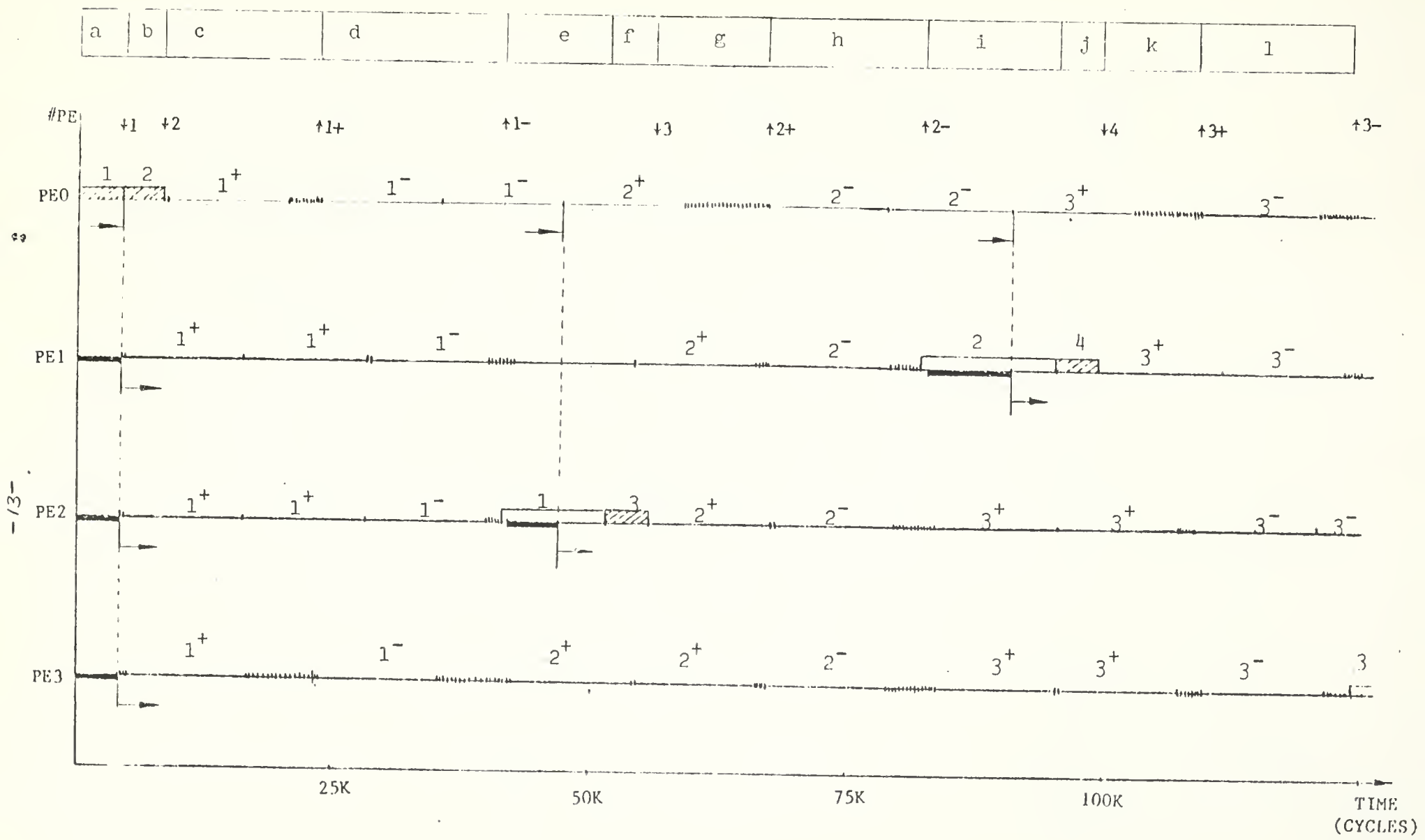
```
Procedure sum(#try)
   update(#try)
   generate(#try+2)
   calculate mult1,mult2:the multiplicities
   calculate parameter lists
repeat suspend(pr=2) until done(#try-1)
request(work,pr=1,mult=mult1,#try+2,paramlist+)
request(work,pr=1,mult=mult2,#try+2,paramlist-)
request(sum ,pr=2,mult=1,#try+2)
   done(#try) := true
   release
end
```

The program is initiated by the following section, which is executed only once:

```
Program initiate
   $ start odd series of tries beginning with #try=1
   generate(#try=1)
   calculate mult1,mult2: the multiplicities
   calculate parameter lists
request(work,pr=1,mult=mult1,#try=1,paramlist+)
request(work,pr=1,mult=mult2,#try=1,paramlist-)
request(sum,pr=2,mult=1    ,#try=1)

   $ start even series of tries beginning with #try=2
   generate(#try=2)
   calculate mult1,mult2: the multiplicities
   calculate parameter lists
request(work,pr=1,mult=mult1,#try=2,paramlist+)
request(work,pr=1,mult=mult2,#try=2,paramlist-)
request(sum,pr=2,mult=1    ,#try=2)
   release
end
```

The time chart at fig. 2.2 shows each processor's execution history during a run with four processors (only a short initial period is shown).


a
b
c
d
e
f
g
h
i
j
k
l
#PE
↑1
↑2
↑1+
↑1-
↑3
↑2+
↑2-
↑4
↑3+
↑3-
PE0
PE1
PE2
PE3
25K
50K
75K
100K
TIME
(CYCLES)
n
parallel work (step 2), try n
short parallel tasks (step 2)
waiting time
generation (step 1) of try n
summary (step 3) of try n

The following is a description of this time chart (see top line for delineation of time intervals):

(a) PE0 generates the first try, the other processors wait for data. When the data are ready they are inserted onto the queue (this time is indicated by ↓1).

(b) PE0 generates the next try, the other processors begin working on the parallel tasks of try1+. When the data for try2 is ready, it is inserted into the queue (↓2).

(c) All the processors work on parallel tasks of try1+. Short tasks consist only of checking whether a "candidate" (a member of a nearby bin) is in fact within r_c. Long parallel tasks include updating of the energy and histogram as well. The deletion of the last task for try1+ from the queue is indicated by ↑1+.

(d) Each processor, upon terminating a previous computation, gets a new task from try1-. The deletion of the last task for try1- is indicated by (↑1-).

(e) PE2 is the first processor that finds no more tasks for try1- in the queue. It gets the summary task for try1 and after a short period of activating this task it busy-waits until all the tasks for try1 are completed. The other processors begin working on the parallel tasks of try2+.

(f) After completing the summary for task1, PE2 generates try3 (the next try in the odd series). When the data are ready, they are inserted onto the queue (↓3).

(g) Parallel computation of try2+.

(h) Parallel computation of try2-.

(i) Summary of try2 by PE1. All other processors begin try3+.

(j) Generation of try4 immediately after the summary of try2 (the even series).

(k) Parallel computation of try3+.

(l) Parallel computation of try3-.

We now wish to give a schematic of the POP version. In this version several insignificant changes from the above scheme were made. The Request to execute tasks is performed by the REQUEST subroutine described above. This routine inserts the tasks onto the list of tasks to be executed. GETTASK takes one of these tasks, if available, and deletes it from the list. When no more tasks are available, the GETTASK routine chooses the alternate RETURN, so that the section immediately following the call GETTASK is skipped. The initial part of the program is combined with the main routine using special values (-1 and 0) for #try.

The schematic program at table 2.1 represents the POP version (in FORTRAN).

```
   #try = 0
C          this part is executed by one processor only (#pe=0).
C          begin initialization of both series.
   IF #pe = 0 THEN
      #try = -1
      GO TO 10
   ENDIF

C          begin work on next #try after synchronization
      1  #try = #try + 1
   CALL SYNCH (1)
   GO TO 20

C          generate new try and request the tasks for it
     10    CALL GENERATE(#try+2)
   calculate mult1,mult2 and parameter lists
C          task+ is the first parallel task
   CALL REQUEST(task+,mult1,#try+2,paramlist+)
C          task- is the second parallel task
   CALL REQUEST(task-,mult2,#try+2,paramlist-)
C          tasksummary is the updating and summary
   CALL REQUEST(tasksummary,mult=1,#try+2)
          data for #try+2 is READY
   IF #try = -1 THEN
      #try = #try + 1
      GO TO 10
   ELSE GO TO 1
   ENDIF

C          check if data ready
     20    WAIT UNTIL data for #try are READY
C          get task and execute until no more tasks left
      2  CALL GETTASK(task+,#try,paramlist),RETURNS(3)
   CALL WORK(#try,paramlist)
   GO TO 2

C          get task and execute until no more tasks left
      3  CALL GETTASK(task-,#try,paramlist),RETURNS(4)
   CALL WORK(#try,paramlist)
   GO TO 3

C          get task if available, otherwise return to the beginning
C          of the loop
      4  CALL GETTASK(tasksummary,#try),RETURNS(1)
   wait until all tasks done for #try
   check if move is accepted
   CALL UPDATE
C          generate next try as soon as this try is updated
   GO TO 10
```

Table 2.1

3. TIMING ANALYSIS.

In timing analysis of the executions of the program the following two extreme cases should be treated differently:

Super-saturated case. There are many more parallel tasks than available PEs. The PE executing the task "summary" waits until all parallel tasks for the corresponding try are completed. (That is the case represented at fig. 2.1).

Under-saturated case. There are many more free PEs than parallel tasks. Most of the PEs are busy-waiting for tasks.

Note that because of the high degree of asynchrony of the program the execution histories are very irregular. Beginning with a super-saturated case and gradually decreasing the total number of PEs in the system (for a fixed size problem) one enters the interval of cases in which the percentage of both types of waiting mentioned above becomes high and they interleave in an irregular way. The stochastic nature of the simulated process itself also adds up to the irregularity. For example when the run is long enough there is a non-zero probability that all particles condense in a single bin.

To simplify our analysis we will freely replace random variables by their mean values and will consider only the two extreme cases (by doing this we reduce somewhat the accuracy of the predictions).

In the super-saturated case let $T_p(m)$ be the average time required to compute m tries using p processors. By analyzing the structure of the computation we come to the following formula:

(3.1) $$p \times T_p(m) = (Tg + Tr + Tp + Tw + Ts + p \times Tsy) \times m$$

where:

$Tg = Tg1 + Tg2 \times c$	is time for generation
$Tr = Tr1 \times 2 \times c \times d + Tr2 \times 2 \times p$	is time for obtaining tasks by requested PEs
$Tp = Tp1 \times 2 \times n + Tp2 \times 2 \times (c \times d - 1)$	is time for parallel task execution
$Ts = Ts1 + (Ts2 + Ts3 \times b) \times a +$ $+ Th \times Nh$	is time for summary execution that includes updating histogram

m is the number of tries.

p is the number of processors.

n is the average number of neighbors (the number of particles within the range of the potential, $r < r_c$).

c is the number of neighbor bins.

d is the average number of particles in each bin.

a is the fraction of accepted moves.

b is the fraction of moves that cross bin boundaries.

Tsy is the average number of instructions spent by one processor by invoking synchronization.

Tg1 is the constant part of the generation.

Tg2 is the neighbor dependent additive to the generation time per bin.

Tr1 is the request overhead time per particle.

Tr2 is the request overhead time per PE.

Tp1 is the time needed for the computation of the potential for a neighbor of a given particle.

Tp2 is the time for checking whether a particle is a neighbor of a given particle or not.

Ts1 is the constant update time for one try.

Ts2 is the additional update time for an accepted try.

Ts3 is the additional update time for an accepted try in case a particle crosses bin boundaries.

Th is the time to update one component of the histogram.

Nh is the number of components in the histogram.

Tw is the average waiting period.

Empirical values for the time constants are:

$$(3.2)\quad \begin{cases} Tg1 = 1450;\ Tg2 = 525;\ Tr1 = 62;\ Tr2 = 54;\ Tp1 = 11730;\ Tp2 = 605; \\ Ts1 = 1200;\ Ts2 = 3060;\ Ts3 = 94;\ Th = 55;\ Tsy = 1503. \end{cases}$$

The values of problem dependent constants for the simulated small size problem are:

$$(3.3)\quad n = 6;\quad c = 3;\quad d = 10;\quad a = 0.7;\quad b = 0.1;\quad Nh = 60.$$

All the time constants whose values are specified in (3.2) correspond to the execution of certain subcodes in the program and are supposed to be data-independent. Their values were obtained by running the program in a special tracing mode, in which the (simulated) time of beginning and of ending of the corresponding subcode were registered, then the former was subtracted from the latter, and the value of the difference was averaged over the execution history with the number of tries, m, equal to 30.

The value of Tw can not be determined in this way and we use the following consideration to deal with it. In the super-saturated case all realizations of Tw are busy-waiting intervals of those processors that got the summary task from the queue but cannot execute the updating included in the summary until all parallel computations corresponding to it are done. In the sustained regime a PE that picked a summary task is the first PE that finds no more parallel task corresponding to this summary (like PE2 at the interval (e) at fig. 2.1). Therefore this PE starts execution of the summary only after all PEs that have been concerned with the potential and histogram computations for this move have completed. Clearly the wait starts no sooner than last initiation of a group of these tasks and ends at the last completion. The time to execute one parallel task by one PE is no more than Tp1 + Tp2. Hence

(3.4) $$Tw \leq Tp1 + Tp2.$$

Since the average value of Tw is not known, we introduce two estimates for $T_p(m)$: $T''_p(m)$ is the value for $T_p(m)$ according to (3.1) with Tw = Tp1 + Tp2; $T'_p(m)$ is the value for $T_p(m)$ according to (3.1) with Tw = 0.

Formula (3.1) with numeric values (3.2) and (3.3) allows us to predict $T'_p(m)$. On the other hand, values $T_p(m)$ could be traced directly. In several examples the predictions of $T'_p(m)$ from (3.1), were compared with the actual values of $T_p(m)$. The predictions were accurate within 8%.

The efficiency $E(p) = T_1(m)/[p \times T_p(m)]$ in the super-saturated case may be estimated as follow. For p = 1 one has Tw = 0 (a single PE can not busy-wait for itself). Hence $T_1(m) = T_1'(m)$. Because of (3.4) one has $T_p(m) \leqslant T_p''(m)$ for any p. Therefore we can accept the following lower bound estimation:

$$E(p) \geqslant T_1'(m) \;/\; [p \times T_p''(m)] \;=_{def}\; e(p).$$

One has $e(p) = 1/(1+\alpha)$, where

$$\alpha = \alpha(p) \;=_{def}\; [Tp1 + Tp2 + (p-1) \times (Tsy + 2 \times Tr2)] \;/\; T_1(1)$$

If $\alpha = 1$, then the parallelization overhead is approximatively 50%. The equation $\alpha(p) = 1$ yields the following upper bound for p:

$$p_o = 1 + (T_1'(1) - Tw) \;/\; (Tsy + 2 \times Tr2)$$

In the simulated example $p_o \approx 111$.

Now we analyze the under-saturated case. Note that all times introduced as terms in formula (3.1) still exist as specific computational interval (for example they may be traced) and the interpretations of all of them except for Tw are still valid. Tw now represents busy-waiting to get tasks for those PEs that execute parallel tasks. Instead of using formula (3.1) (which is still valid but does not predict $T_p(m)$ because now Tw is not restricted) we use the following consideration.

The critical path for the execution of one particle move is: 1) execution of the synchronization call, Tsy; 2) generation, Tg; 3) execution all (+) tasks and then execution all (-) tasks, $2 \times (Tp1 + Tr1)$; 4) summary, Ts.

Since two tries may execute concurrently the average time spent by one particle in the under-saturated regime is $t_0 = [Tg + Ts + 2\times(Tp1 + Tp2 + 2\times Tr1)]/2$. During time interval t_0 all parallel tasks of one try can be completed. If $p \geq 2$ is the total number of the PEs, then 2 PEs are working on the critical path and p-2 PEs must complete an amount of work no more than $A = Tp + Tr + Tsy\times(p-2)$ during time no less than t_0, i.e. $(p-2)\times t_0 \geq A$. Hence we have the following critical value $p = p_s$ for the number of PEs in the supersaturated case:

$$p_s = 2\times[1 + (Tp+Tr)/((2[Tp1+Tp2+Tr1-Tsy)+Tg+Ts)]$$

In the simulated example $p_s = 14$.

Running the problem with $1 < p < MIN\{p_o, p_s\}$ is quite efficient since the problem is super saturated and the parallelization overhead is less than the useful computation. If $p_o < p_s$, we can choose $p_o < p < p_s$, decreasing the total computation time $T_p(m)$, but lowering the efficiency below 50%. Choosing $p > p_s$ does not lower the computation time.

According to this analysis we can predict the efficiency for a complex three dimensional problem. We distinguish between "neighbors" i.e. particles which are within a distance r_c of a given particle (and for which the possibly time consuming potential calculation must be carried through), and "candidates" which are particles in neighboring subregions which must be examined to see if they are neighbors. Assuming a three-dimensional problem with 15000 neighbors out of 30000

candidates distributed in 27 bins (c=2,7, n=15000, d=1111, a=0.99, b=0.1, Nh=60), we have:

$p_s \approx 17532$ $p_o \approx 243286$

with efficiency low bounds e(p):

e(2048) = 99.2%

e(4096) = 98.3%

e(8192) = 96.7%

e(16384) = 93.7%

In a small problem with 65 neighbors (c=27, n=65, d=4.8, a=0.99, b=0.1, Nh=60) we have:

$p_s = 78$ $p_o = 1062$

and the efficiencies are:

e(16) = 97.9%

e(32) = 96.5%

e(64) = 93.8%

e(128) = 88.8%

e(256) = 80.2%

e(512) = 67.3%

4. CONCLUSION

We conclude that the strategy employed here will be very effective for carrying out very large Monte Carlo calculations of the eqilibrium properties of complex systems. In circumstances where one can confidently expect that other users are present, it may be unnecessary to use the tactic of interleaving odd even tries. As it stands the method seems effective enough to consider using for all problems. That is simulations that one could carry out by constructing multiple replicas and assigning each to one or a few processors might be better treated by simulating a physically large system and avoiding boundary effects.

Finally we would like to remark that another widely used molecular simulation method, solution of Newton equations under the influence of the assumed forces, can be treated in much the same way. For this algorithm, pair sums (of the forces) must be computed and these can be made parallel as we have done. Neighbor enumeration methods work exactly as here. The difference is that the remaining work can also be made parallel. That is, the integration of the equations of motion for each particle, given the forces on the particle, can be carried out in separate processors.

REFERENCES

[1] N. Metropolis, A. W. Rosenbluth, N. M. Rosenbluth, A. M. Teller, E. Teller, J. Chem. Phys., vol.21, p.1087 (1953).

[2] Allan Gottlieb, MOP - A Multiprocessor Operating System Extending WASHCLOTH, Ultracomputer Note #13, Courant Institute, N.Y.U., 1980.

[3] Allan Gottlieb, B. D. Lubachevsky and Larry Rudolph, Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors, Ultracomputer Note #16, Courant Institute, N.Y.U., 1980.

[4] Allan Gottlieb and Clyde P. Kruskal. Coordinating Parallel Processors: A Partial Unification. Ultracomputer Note #34. (September, 1981). Courant Institute, NYU.

[5] J. T. Schwartz. Ultracomputers. ACM Trans. on Prog. Languages and Systems, Vol.2, No.4 (October 1980), 484-521.

[6] Allan Gottlieb, Washcloth - The Logical Successor to Soapsuds, Ultracomputer Note #12, Courant Institute, N.Y.U., 1980.